IIT KHARAGPUR

NPTEL ONLINE
CERTIFICATION COURSES

Lecture 30: MODELING MEMORY

PROF. INDRANIL SENGUPTA

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

How to Model Memory?

- Memory is typically included by instantiating a pre-designed module from a design library.
- Alternatively, we can model memories using two-dimensional arrays.
  - Array of register variables (behavioral model).
  - Mainly used for simulation purposes.
  - Even used for the synthesis of small-size memories.

Typical Example

```
module  memory_model ( …….. )
   …
   reg [7:0] mem [0:1023];
   …
endmodule
```

```
module  memory_model ( …….. )
   reg [7:0] mem [0:1023];
   initial  begin
     mem[0] = 8'b01001101;
     mem[4] = 8'b00000000;
   end
endmodule
```

- Each memory word is of type [7:0], i.e. 8 bits.
- The memory words can be accessed as **mem[0]**, **mem[1]**, …, **mem[1023]**.

How to Initialize Memory?

- By reading memory data patterns from a specified disk file.
  - Used for simulation.
  - Used in test benches.
- Two Verilog functions can be used:

  `$readmemb (filename, memname, startaddr, stopaddr)`

  (Data is read in binary format)

  `$readmemh (filename, memname, startaddr, stopaddr)`

  (Data is read in hexadecimal format)

  - If "*startaddr*" and "*stopaddr*" are omitted, the entire memory is read.

Example 1: Initializing a memory from file

```
module memory_model ( …….. );
  reg [7:0]  mem[0:1023];
  initial
    begin
     $readmemh (“mem.dat”, mem);
    end
endmodule
```

```
module memory_model ( …….. );
  reg [7:0]  mem[0:1023];
  initial
    begin
     $readmemb (“mem.dat”, mem,
                  200, 50);
    end
endmodule
```

Example 2: Single-port RAM with synchronous read/write

```
module  ram_1 (addr, data, clk, rd, wr, cs);
  input  [9:0] addr;    input  clk, rd, wr, cs;
  inout  [7:0] data;
  reg [7:0]  mem [1023:0];    reg [7:0] d_out;

  assign  data = (cs && rd) ? d_out : 8'bz;
  always  @(posedge clk)
    if (cs && wr && !rd)  mem[addr] = data;
  always  @(posedge clk)
    if (cs && rd && !wr)  d_out = mem[addr];
endmodule
```

Example 3: Single-port RAM with asynchronous read/write

```
module  ram_2 (addr, data, rd, wr, cs);
  input  [9:0] addr;    input  rd, wr, cs;
  inout  [7:0] data;
  reg [7:0]  mem[1023:0];    reg [7:0] d_out;

  assign  data = (cs && rd) ? d_out : 8'bz;
  always  @(addr or data or rd or wr or cs)
     if (cs && wr && !rd)  mem[addr] = data;
  always  @(addr or rd or wr or cs)
     if (cs && rd && !wr)  d_out = mem[addr];
endmodule
```

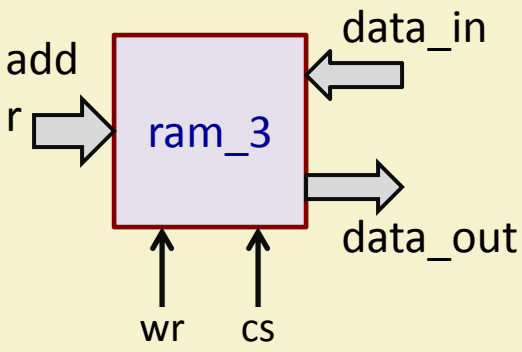
Example 4: A ROM / EPROM

```
module  rom (addr, data, rd_en, cs);
   input  [2:0] addr;    input  rd_en, cs;
   output reg [7:0] data;
   always  @(addr or rd_en or cs)
        case (addr)
           0:   data = 22;
           1:   data = 45;
            ……..……
           7:   data = 12;
        endcase
 endmodule
```

An Important Point to Note

- Some simulation or synthesis tools give inconsistent behavior when using the "*inout*" data type.
  - Such "*inout*" bidirectional data should be avoided.
- A better way to design a memory unit is to keep the data input and data output bus signal lines separate.
  - An example memory description with separate data buses is shown on the next slide.

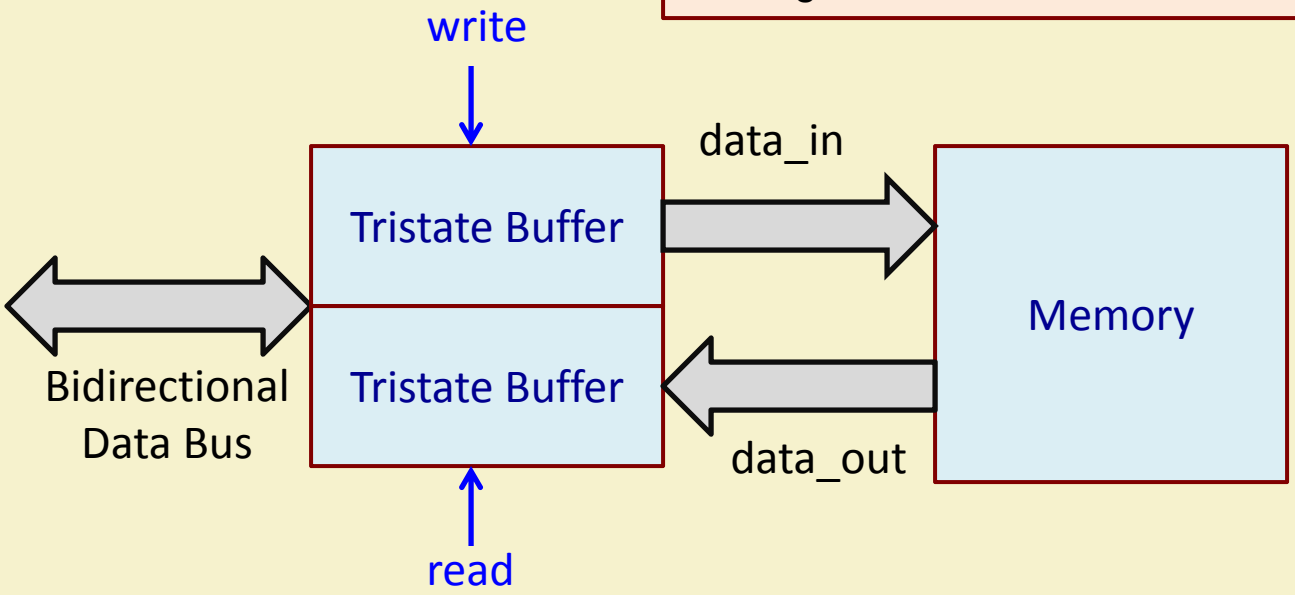
Example 4




```
module  ram_3 (data_out, data_in, addr, wr, cs);
  parameter addr_size = 10, word_size = 8,
            memory_size = 1024;
  input  [addr_size-1:0] addr;
  input  [word_size-1:0] data_in;
  input  wr, cs;
  output [word_size-1:0] data_out;
  reg [word_size-1:0]  mem [memory_size-1:0];

  assign data_out = mem[addr];

  always  @(wr or cs)
     if (wr)  mem[addr] = data_in;
endmodule
```

- For bidirectional data bus, tristate buffers can be included explicitly.

```
tri [7:0] Bus;
wire [7:0] data_out, data_in;
assign Bus = read ? data_out : 8'hzz;
assign data_in = write ? Bus : 8'hzz;
```



Simple Test Bench using ram_3

```
module RAM_test;
  reg [9:0] address;
  wite [7:0] data_out;
  reg [7:0] data_in;
  reg write, select;
  integer k, myseed;

  ram_3 RAM (data_out, data_in, address, write, select);

  initial
    begin
      for (k=0; k<=1023; k=k+1)
        begin
          address = k;
          data = (k + k) % 256; read = 0; write = 1; select = 1;
          #2 write = 0; select = 0;
        end
```

```
      repeat (20)
       begin
         #2 address = $random(myseed) % 1024;
         write = 0; select = 1;
         $display ("Address: %5d, Data: %4d", address,
                    data);
         #2 select = 0;
       end
    end

  initial  myseed = 35;
endmodule
```

```
Address:    0, Data:   0
Address:  960, Data: 128
Address:  482, Data: 196
Address:  693, Data: 106
Address:  246, Data: 236
Address:  228, Data: 200
Address:  148, Data:  40
Address:  767, Data: 254
Address:  355, Data: 198
Address:  259, Data:   6
Address:   21, Data:  42
Address:  872, Data: 208
Address:  758, Data: 236
Address:  193, Data: 130
Address:  909, Data:  26
Address:  632, Data: 240
Address:  719, Data: 158
Address:  214, Data: 172
Address:   67, Data: 134
Address:  908, Data:  24
```

END OF LECTURE 30

IIT KHARAGPUR

NPTEL ONLINE
CERTIFICATION COURSES

Lecture 31: MODELING REGISTER BANKS

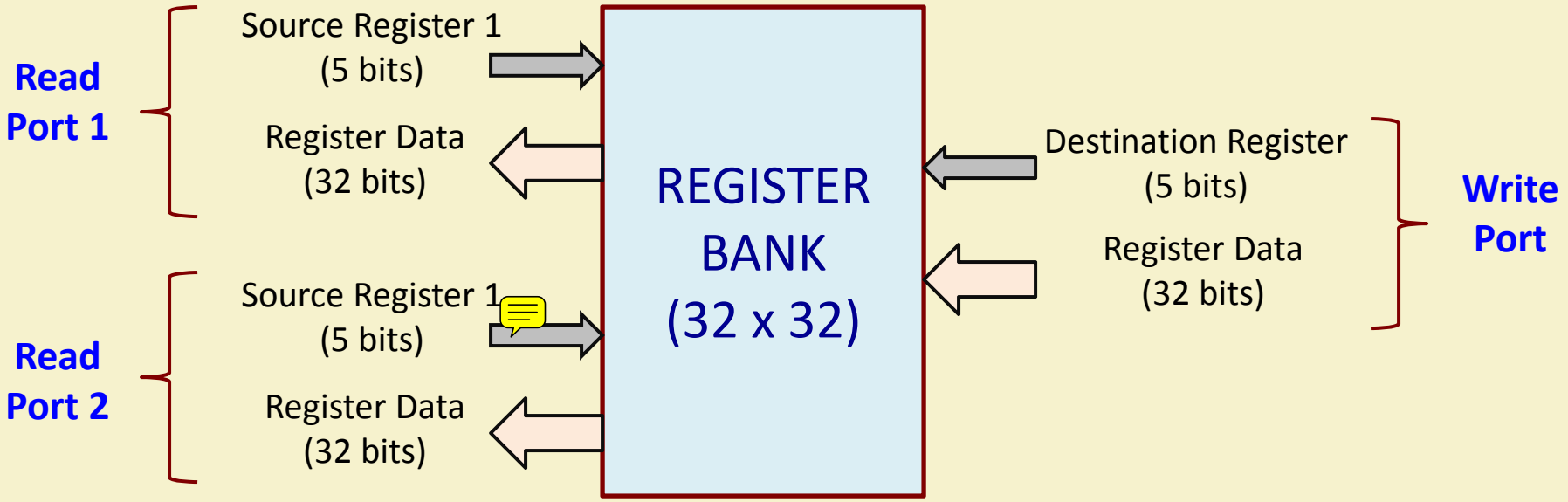
PROF. INDRANIL SENGUPTA

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Introduction

- A register bank or register file is a group of registers, any of which can be randomly accessed.
  - Commonly used in computers to store the user-accessible registers.
  - For example, in the MIPS32 processor, there are 32 32-bit registers, referred to as *R0*, *R1*, …, *R31*.
- Can be implemented in Verilog as independent registers, or as an array of registers similar to a memory.
- Registers banks often allow concurrent accesses.
  - MIPS32 allows *2 register reads and 1 register write every clock cycle*.

- We show various ways of designing a register bank that supports two reads and one write simultaneously.
  - It is assumed that the same register is not read and written simultaneously.


Read Port 1
Source Register 1 (5 bits)
Register Data (32 bits)
Read Port 2
Source Register 1 (5 bits)
Register Data (32 bits)
REGISTER BANK (32 x 32)
Destination Register (5 bits)
Register Data (32 bits)
Write Port

```
// 4 x 32 register file
module regbank_v1 (rdData1, rdData2, wrData, sr1, sr2, dr, write, clk);
  input clk, write;
  input [1:0] sr1, sr2, dr;  // Source and destination registers
  input [31:0] wrData;
  output reg [31:0] rdData1, rdData2;
  reg [31:0] R0, R1, R2, R3;

  always @(*)
    begin
      case (sr1)
        0: rdData1 = R0;
        1: rdData1 = R1;
        2: rdData1 = R2;
        3: rdData1 = R3;
        default: rdData1 = 32'hxxxxxxxx;
      endcase
  end
```

```
  always @(*)
    begin
      case (sr2)
        0: rdData2 = R0;
        1: rdData2 = R1;
        2: rdData2 = R2;
        3: rdData2 = R3;
        default: rdData2 = 32'hxxxxxxxx;
      endcase
  end
  always @(posedge clk)
    begin
      if (write)
        case (dr)
          0: R0 <= wrData;
          1: R1 <= wrData;
          2: R2 <= wrData;
          3: R3 <= wrData;
        endcase
    end
endmodule
```

This way of modeling is feasible if the number of registers is small.

```
// 4 x 32 register file
module regbank_v2 (rdData1, rdData2, wrData, sr1, sr2, dr, write, clk);
  input clk, write;
  input [1:0] sr1, sr2, dr;  // Source and destination registers
  input [31:0] wrData;
  output [31:0] rdData1, rdData2;
  reg [31:0] R0, R1, R2, R3;

  assign rdData1 = (sr1 == 0) ? R0 :
                   (sr1 == 1) ? R1 :
                   (sr1 == 2) ? R2 :
                   (sr1 == 3) ? R3 : 0;
  assign rdData2 = (sr2 == 0) ? R0 :
                   (sr2 == 1) ? R1 :
                   (sr2 == 2) ? R2 :
                   (sr2 == 3) ? R3 : 0;
```

```
always @(posedge clk)
    begin
      if (write)
        case (dr)
          0: R0 <= wrData;
          1: R1 <= wrData;
          2: R2 <= wrData;
          3: R3 <= wrData;
        endcase
    end
endmodule
```

```
// 32 x 32 register file
module regbank_v3 (rdData1, rdData2, wrData, sr1, sr2, dr, write, clk);
  input clk, write;
  input [4:0] sr1, sr2, dr;  // Source and destination registers
  input [31:0] wrData;
  output [31:0] rdData1, rdData2;

  reg [31:0] regfile[0:31];

  assign rdData1 = regfile[sr1];
  assign rdData2 = regfile[sr2];

  always @(posedge clk)
    if (write) regfile[dr] <= wrData;
endmodule
```

```
// 32 x 32 register file with reset facility
module regbank_v4 (rdData1, rdData2, wrData, sr1, sr2, dr, write, reset, clk);
  input clk, write, reset;
  input [4:0] sr1, sr2, dr;    // Source and destination registers
  input [31:0] wrData;
  output [31:0] rdData1, rdData2;
  integer k;

  reg [31:0] regfile[0:31];

  assign rdData1 = regfile [sr1];
  assign rdData2 = regfile [sr2];
```

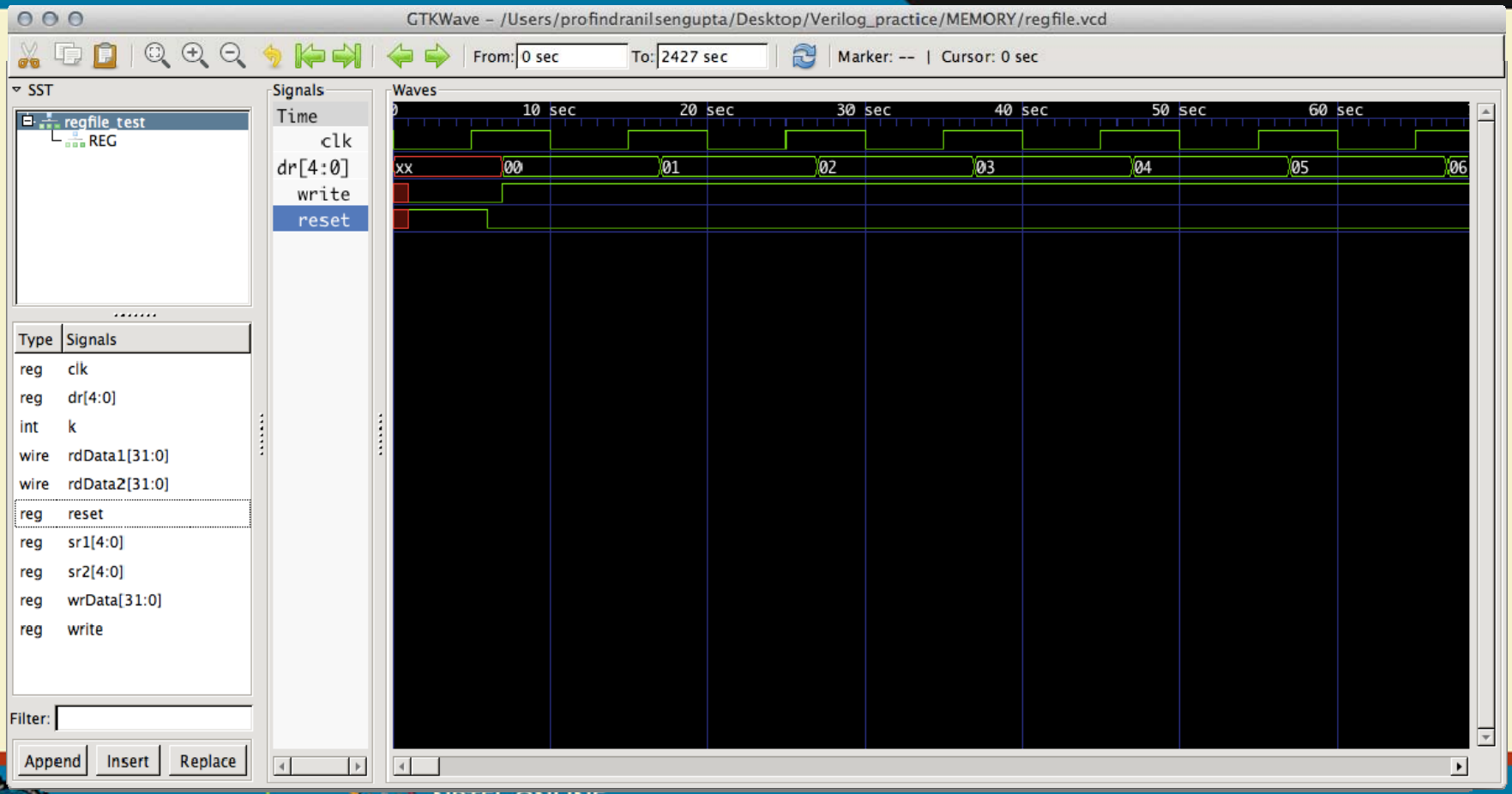
```
always @(posedge clk)
    begin
      if (reset) begin
        for (k=0; k<32; k=k+1) begin
          regfile[k] <= 0;
        end
      end
      else begin
             if (write)
                regfile[dr] <= wrData;
          end
    end
endmodule
```

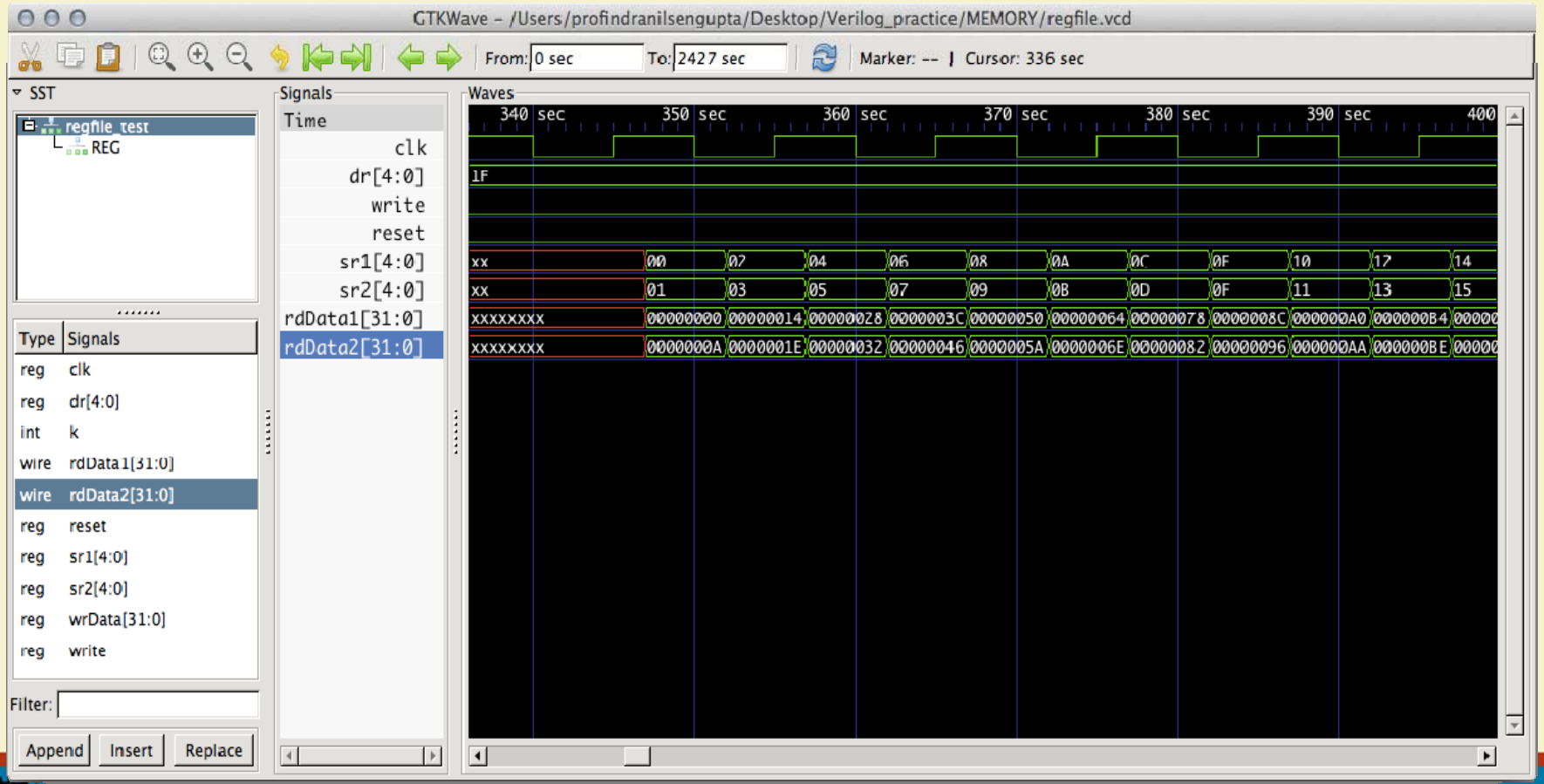
A test bench to verify operation of the register file

```
module regfile_test;

reg [4:0] sr1, sr2, dr;
reg[31:0] wrData;
reg write, reset, clk;
wire [31:0] rdData1, rdData2;
integer k;

regbank_v4 REG (rdData1, rdData2, wrData, sr1, sr2, dr, write, reset, clk);

initial clk = 0;

always #5 clk = !clk;

initial
  begin
    $dumpfile ("regfile.vcd");  $dumpvars (0, regfile_test);
    #1 reset = 1; write = 0;
    #5 reset = 0;
  end
```

```
initial
  begin
    #7
    for (k=0; k<32; k=k+1)
      begin
        dr = k; wrData = 10* k; write = 1;
        #10 write = 0;
      end

    #20
    for (k=0; k<32; k=k+2)
      begin
        sr1 = k; sr2 = k+1;
        #5;
        $display ("reg[%2d] = %d, reg[%2d] = %d", sr1, rdData1, sr2, rdData2);
      end

     #2000 $finish;
  end
endmodule
```

```
reg[ 0] =          0, reg[ 1] =         10
reg[ 2] =         20, reg[ 3] =         30
reg[ 4] =         40, reg[ 5] =         50
reg[ 6] =         60, reg[ 7] =         70
reg[ 8] =         80, reg[ 9] =         90
reg[10] =        100, reg[11] =        110
reg[12] =        120, reg[13] =        130
reg[14] =        140, reg[15] =        150
reg[16] =        160, reg[17] =        170
reg[18] =        180, reg[19] =        190
reg[20] =        200, reg[21] =        210
reg[22] =        220, reg[23] =        230
reg[24] =        240, reg[25] =        250
reg[26] =        260, reg[27] =        270
reg[28] =        280, reg[29] =        290
reg[30] =        300, reg[31] =        310
```


GTKWave - /Users/profindranilsengupta/Desktop/Verilog_practice/MEMORY/regfile.vcd
From: 0 sec
To: 2427 sec
Marker: -- | Cursor: 0 sec
SST
regfile_test
REG
Type Signals
reg clk
reg dr[4:0]
int k
wire rdData1[31:0]
wire rdData2[31:0]
reg reset
reg sr1[4:0]
reg sr2[4:0]
reg wrData[31:0]
reg write
Filter:
Append
Insert
Replace
Signals
Time
clk
dr[4:0]
write
reset
Waves
10 sec
20 sec
30 sec
40 sec
50 sec
60 sec
xx
00
01
02
03
04
05
06


GTKWave - /Users/profindranilsengupta/Desktop/Verilog_practice/MEMORY/regfile.vcd
From: 0 sec
To: 2427 sec
Marker: -- | Cursor: 336 sec
SST
regfile_test
REG
Type Signals
reg clk
reg dr[4:0]
int k
wire rdData1[31:0]
wire rdData2[31:0]
reg reset
reg sr1[4:0]
reg sr2[4:0]
reg wrData[31:0]
reg write
Filter:
Append
Insert
Replace
Signals
Time
clk
dr[4:0]
write
reset
sr1[4:0]
sr2[4:0]
rdData1[31:0]
rdData2[31:0]
Waves
340 sec
350 sec
360 sec
370 sec
380 sec
390 sec
400

END OF LECTURE 31

IIT KHARAGPUR

NPTEL ONLINE
CERTIFICATION COURSES

Lecture 32: BASIC PIPELINING CONCEPTS

PROF. INDRANIL SENGUPTA

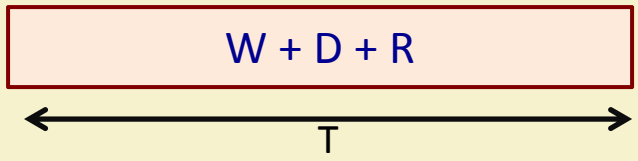
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

What is Pipelining?

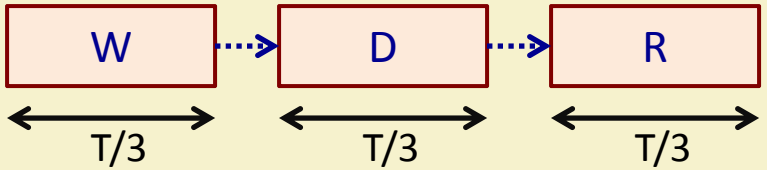
- A mechanism for overlapped execution of several input sets by partitioning some computation into a set of *k* sub-computations (or stages).
  - Very nominal increase in the cost of implementation.
  - Very significant speedup (ideally, *k*).
- Where are pipelining used in a computer system?
  - **Instruction execution**: Several instructions executed in some sequence.
  - **Arithmetic computation**: Same operation carried out on several data sets.
  - **Memory access**: Several memory accesses to consecutive locations are made.

A Real-life Example

- Suppose you have built a machine *M* that can wash (*W*), dry (*D*), and iron (*R*) clothes, one cloth at a time.
  - Total time required is *T*.
- As an alternative, we split the machine into three smaller machines M_W, M_D and M_R, which can perform the specific task only.
  - Time required by each of the smaller machines is *T/3* (say).

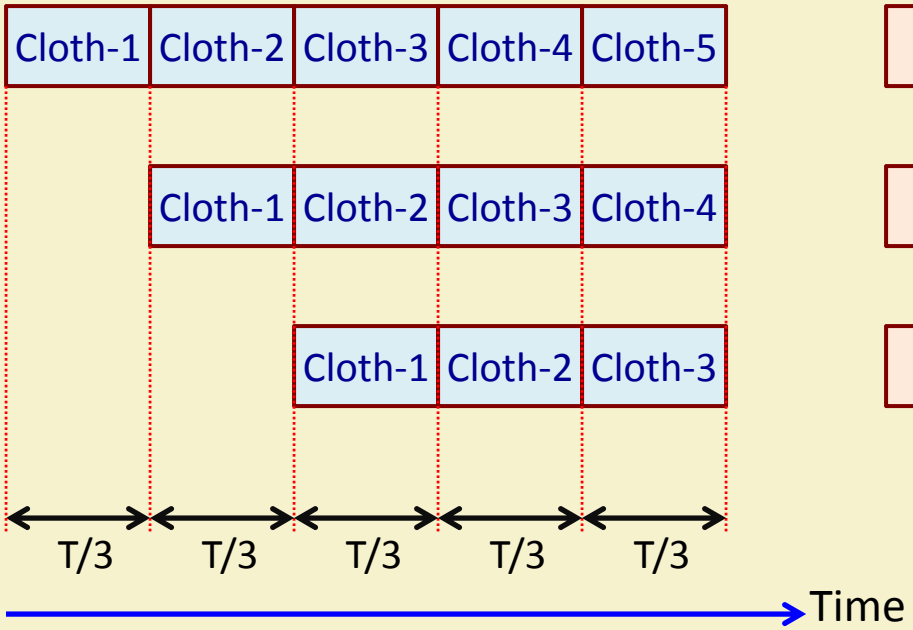


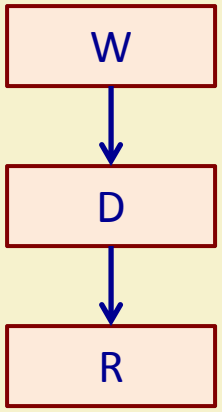
For N clothes, time $T_1 = N.T$



For N clothes, time $T_3 = (2 + N).T/3$

How does the pipeline work?





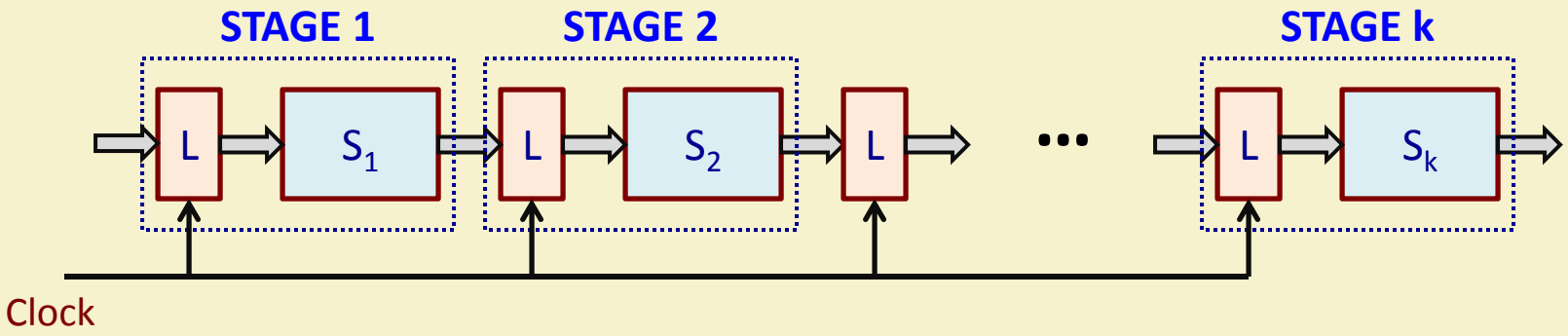
Finishing times:

- Cloth-1 – 3.T/3
- Cloth-2 – 4.T/3
- Cloth-3 – 5.T/3
- …
- Cloth-N – (2 + N).T/3

Extending the Concept to Processor Pipeline

- The same concept can be extended to hardware pipelines.
- Suppose we want to attain *k* times speedup for some computation.
  - **Alternative 1**: Replicate the hardware *k* times → cost also goes up *k* times.
  - **Alternative 2**: Split the computation into *k* stages → very nominal cost increase.
- Need for buffering:
  - In the washing example, we need a tray between machines (W & D, and D & R) to keep the cloth temporarily before it is accepted by the next machine.
  - Similarly in hardware pipeline, we need a *latch* between successive stages to hold the intermediate results temporarily.

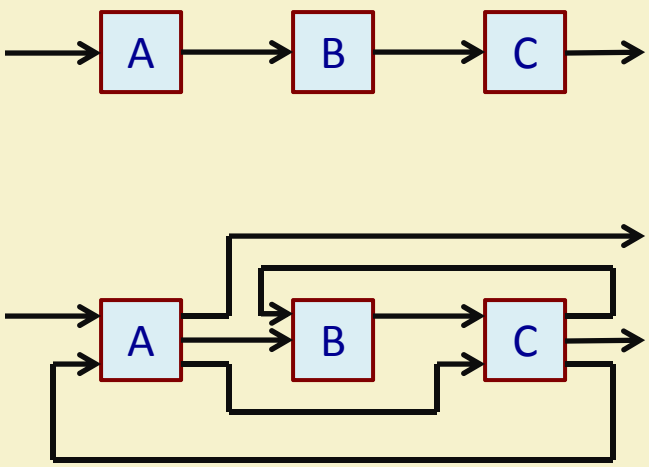
Model of a Synchronous k-stage Pipeline



- The latches are made with master-slave flip-flops, and serve the purpose of isolating inputs from outputs.
- The pipeline stages are typically combinational circuits.
- When *Clock* is applied, all latches transfer data to the next stage simultaneously.

Structure of the Pipeline

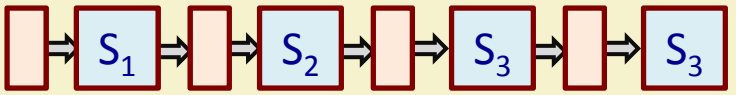
- **Linear Pipeline**: The stages that constitute the pipeline are executed one by one in sequence (say, from left to right).
- **Non-linear Pipeline**: The stages may not execute in a linear sequence (say, a stage may execute more than once for a given data set).



A possible sequence: A, B, C, B, C, A, C, A

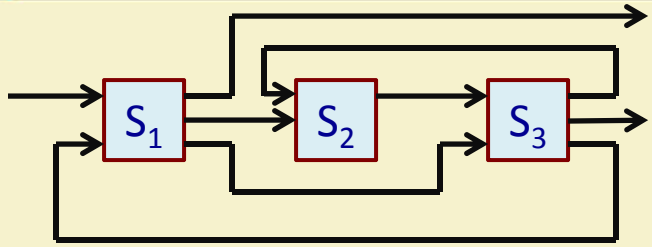
Reservation Table

- The *Reservation Table* is a data structure that represents the utilization pattern of successive stages in a synchronous pipeline.
  - Basically a space-time diagram of the pipeline that shows precedence relationships among pipeline stages.
    - X-axis shows the time steps
    - Y-axis shows the stages
  - Number of columns give evaluation time.
  - The reservation table for a 4-stage linear pipeline is shown.



	1	2	3	4
S_1	X			
S_2		X		
S_3			X	
S_4				X

- Reservation table for a 3-stage dynamic multi-function pipeline is shown.
  - Contains feedforward and feedback connections.
  - Two functions X and Y.
- Some characteristics:
  - *Multiple X's in a row* :: repeated use of the same stage in different cycles.
  - *Contiguous X's in a row* :: extended use of a stage over more than one cycles.
  - *Multiple X's in a column* :: multiple stages are used in parallel during a clock cycle.



	1	2	3	4	5	6	7	8
S_1	X					X		X
S_2		X		X				
S_3			X		X		X	

	1	2	3	4	5	6
S_1	Y				Y	
S_2			Y			
S_3		Y		Y		Y

Speedup and Efficiency

Some notations:

τ :: clock period of the pipeline

t_i :: time delay of the circuitry in stage S_i

d_L :: delay of a latch

Maximum stage delay $\tau_m = \max \{t_i\}$

Thus, $\tau = \tau_m + d_L$

Pipeline frequency $f = 1 / \tau$

– If one result is expected to come out of the pipeline every clock cycle, f will represent the maximum throughput of the pipeline.

- The total time to process *N* data sets is given by

$$T_k = [(k-1) + N].\tau$$

(k – 1) τ time required to fill the pipeline
1 result every τ time after that → total N.τ

- For an equivalent non-pipelined processor (i.e. one stage), the total time is

$$T_1 = N.k.\tau$$

(ignoring the latch overheads)

- Speedup of the k-stage pipeline over the equivalent non-pipelined processor:

$$S_k = \frac{T_1}{T_k} = \frac{N.k.\tau}{k.\tau + (N-1).\tau} = \frac{N.k}{k + (N-1)}$$

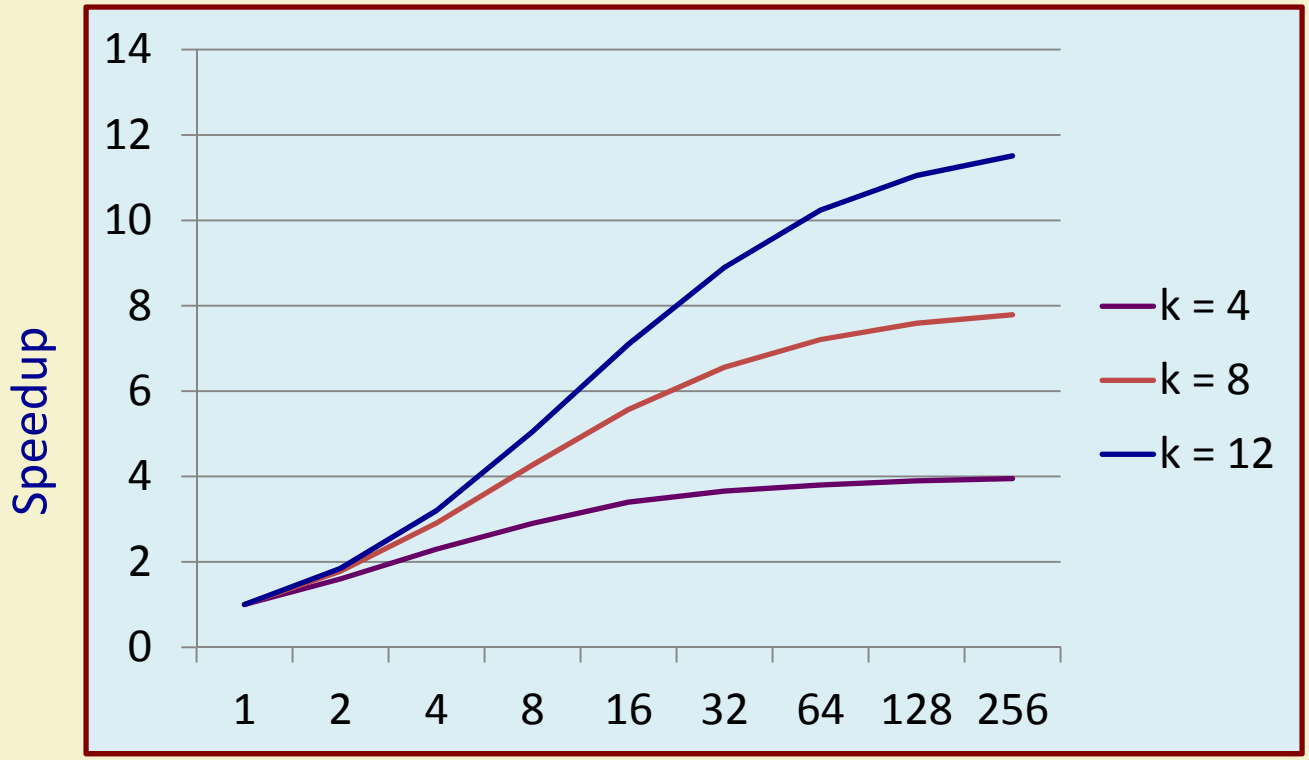
As $N \rightarrow \infty$, $S_k \rightarrow k$

- Pipeline efficiency:
  - How close is the performance to its ideal value?

$$E_k = \frac{S_k}{k} = \frac{N}{k + (N - 1)}$$

- Pipeline throughput:
  - Number of operations completed per unit time.

$$H_k = \frac{N}{T_k} = \frac{N}{[k + (N - 1)].\tau}$$


Speedup
14
12
10
8
6
4
2
0
1
2
4
8
16
32
64
128
256
k = 4
k = 8
k = 12
Number of tasks
N

Clock Skew / Jitter / Setup time

- The minimum clock period of the pipeline must satisfy the inequality:

$$\tau \geq t_{skew+jitter} + t_{logic+setup}$$

- Definitions:
  - *Skew*: Maximum delay difference between the arrival of clock signals at the stage latches.
  - *Jitter*: Maximum delay difference between the arrival of clock signal at the same latch.
  - *Logic delay*: Maximum delay of the slowest stage in the pipeline.
  - *Setup time*: Minimum time a signal needs to be stable at the input of a latch before it can be captured.

END OF LECTURE 32

IIT KHARAGPUR

NPTEL ONLINE
CERTIFICATION COURSES

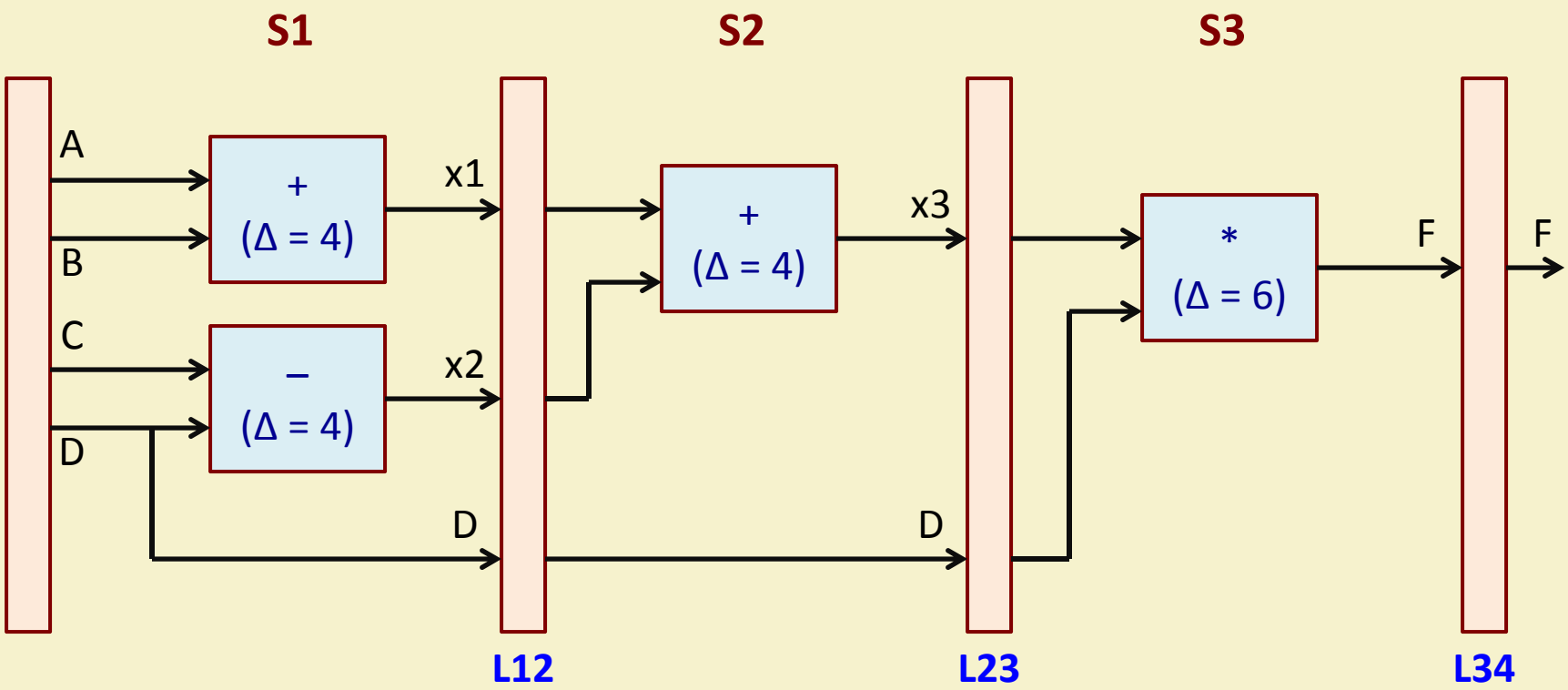
Lecture 33: PIPELINE MODELING (PART 1)

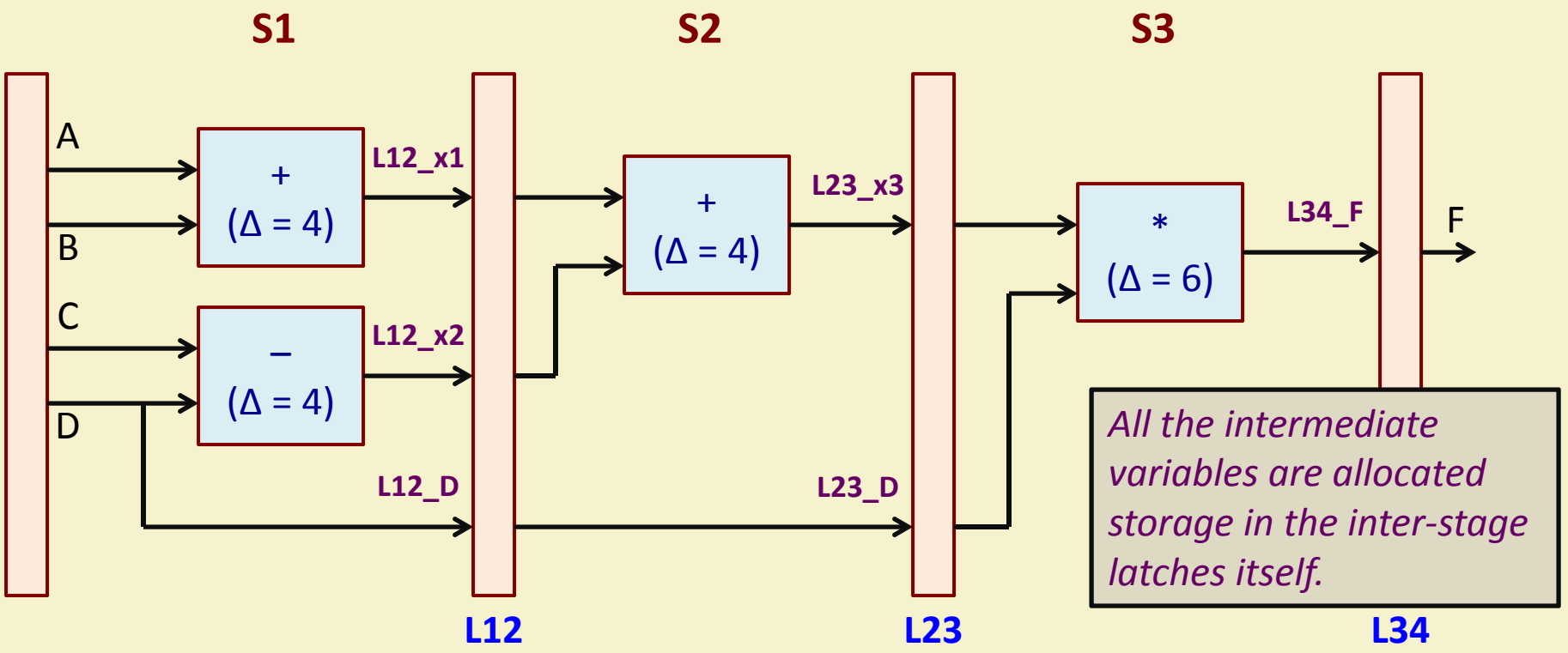
PROF. INDRANIL SENGUPTA

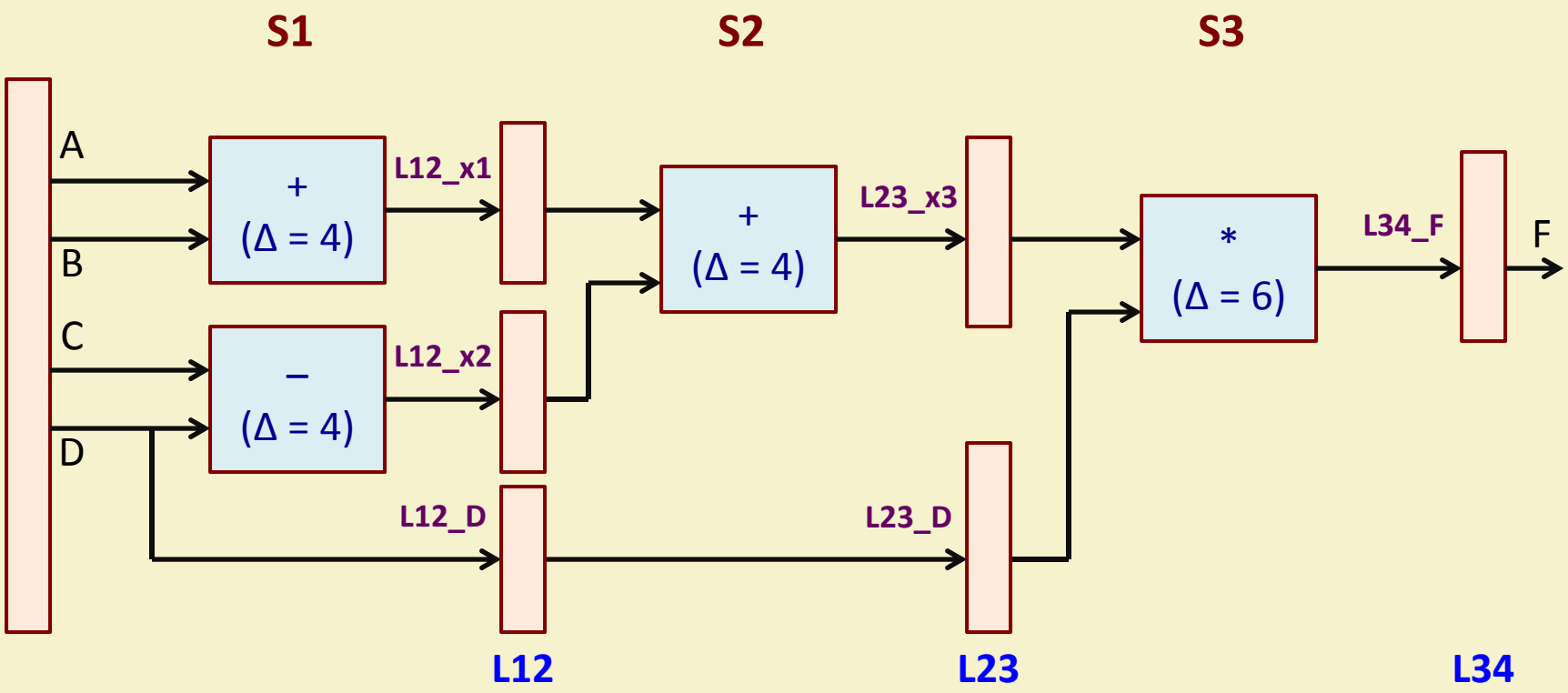
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

A Simple Example

- We consider the example of a very simple 3-stage pipeline.
  - Four *N*-bit unsigned integers *A*, *B*, *C* and *D* as inputs.
  - An *N*-bit unsigned integer *F* as output.
  - The following computations are carried out in the stages:
    a) *S1:* *x1 = A + B; x2 = C – D;*
    b) *S2:* *x3 = x1 + x2;*
    c) *S3:* *F = x3 * D;*
  - Point to note:
    - Input *D* is used in *S1* as well as *S3*.
    - So the value of *D* must be forwarded to *S2* and then to *S3*.


S1
S2
S3
A
B
C
D
+
(Δ = 4)
–
(Δ = 4)
x1
x2
D
+
(Δ = 4)
x3
D
*
(Δ = 6)
F
F
L12
L23
L34


S1
S2
S3
A
B
C
D
+
(Δ = 4)
–
(Δ = 4)
L12_x1
L12_x2
L12_D
+
(Δ = 4)
L23_x3
L23_D
*
(Δ = 6)
L34_F
F
L12
L23
L34
All the intermediate variables are allocated storage in the inter-stage latches itself.


S1
S2
S3
A
B
C
D
+
(Δ = 4)
–
(Δ = 4)
+
(Δ = 4)
*
(Δ = 6)
L12_x1
L12_x2
L12_D
L23_x3
L23_D
L34_F
F
L12
L23
L34

Pipeline Modeling

```
module pipe_ex (F, A, B, C, D, clk);
  parameter N = 10;

  input [N-1:0] A, B, C, D;
  input clk;
  output [N-1:0] F;
  reg [N-1:0] L12_x1, L12_x2, L12_D, L23_x3, L23_D, L34_F;

  assign F = L34_F;

  always @(posedge clk)
    begin
      L12_x1 <= #4 A + B;
      L12_x2 <= #4 C - D;
      L12_D  <= D;                  // ** STAGE 1 **
```

```
    L23_x3 <= #4 L12_x1 + L12_x2;
    L23_D  <= L12_D;                // ** STAGE 2 **

    L34_F  <= #6 L23_x3 * L23_D;    // ** STAGE 3 **
  end

endmodule
```

```
module pipe_ex (F, A, B, C, D, clk);
  parameter N = 10;

  input [N-1:0] A, B, C, D;
  input clk;
  output [N-1:0] F;
  reg [N-1:0] L12_x1, L12_x2, L12_D, L23_x3, L23_D, L34_F;

  assign F = L34_F;

  always @(posedge clk)                    // ** STAGE 1 **
    begin
      L12_x1 <= #4 A + B;
      L12_x2 <= #4 C - D;
      L12_D  <= D;
    end
```

Alternate way of coding:

- One stage per “always” block.
- Code is more readable.

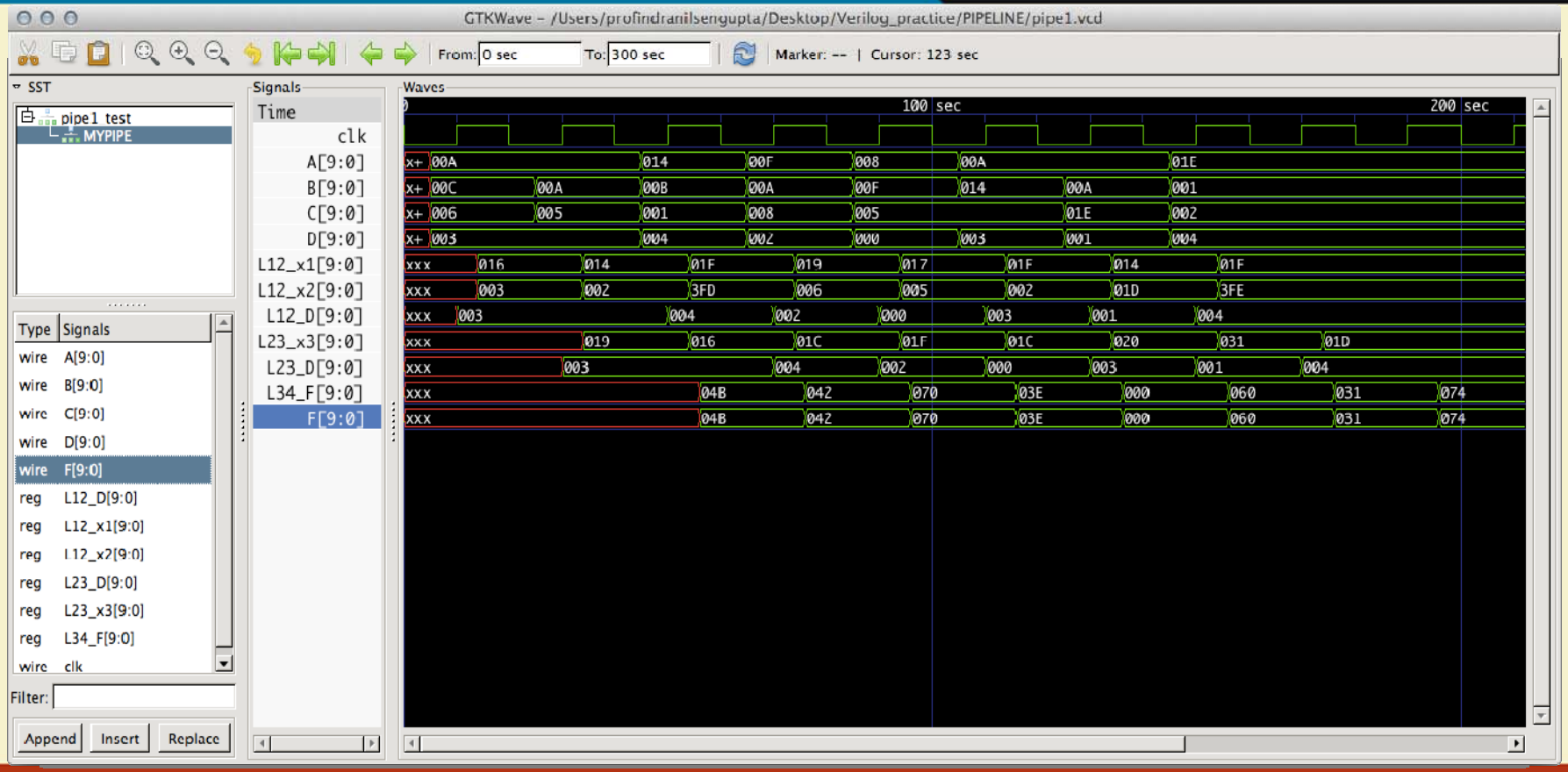
```
  always @(posedge clk)                    // ** STAGE 2 **
    begin
      L23_x3 <= #4 L12_x1 + L12_x2;
      L23_D  <= L12_D;
    end

  always @(posedge clk)                    // ** STAGE 3 **
      L34_F  <= #6 L23_x3 * L23_D;

endmodule
```

```
module pipe1_test;

  parameter N = 10;
  wire [N-1:0] F;
  reg [N-1:0] A, B, C, D;
  reg clk;

  pipe_ex MYPIPE (F, A, B, C, D, clk);

  initial  clk = 0;

  always  #10 clk = ~clk;
```

Pipeline Test Bench

```
  initial
    begin
      #5  A = 10; B = 12; C = 6;  D = 3;  // F = 75   (4Bh)
      #20 A = 10; B = 10; C = 5;  D = 3;  // F = 66   (42h)
      #20 A = 20; B = 11; C = 1;  D = 4;  // F = 112  (70h)
      #20 A = 15; B = 10; C = 8;  D = 2;  // F = 62   (3Eh)
      #20 A = 8;  B = 15; C = 5;  D = 0;  // F = 0    (00h)
      #20 A = 10; B = 20; C = 5;  D = 3;  // F = 66   (42h)
      #20 A = 10; B = 10; C = 30; D = 1;  // F = 49   (31h)
      #20 A = 30; B = 1;  C = 2;  D = 4;  // F = 116  (74h)
    end

  initial
    begin
      $dumpfile ("pipe1.vcd");
      $dumpvars (0, pipe1_test);
      $monitor ("Time: %d, F = %d", $time, F);
      #300 $finish;
    end
endmodule
```

```
Time:     0, F =    x
Time:    56, F =   75
Time:    76, F =   66
Time:    96, F =  112
Time:   116, F =   62
Time:   136, F =    0
Time:   156, F =   96
Time:   179, F =   49
Time:   196, F =  116
```


GTKWave - /Users/profindranilsengupta/Desktop/Verilog_practice/PIPELINE/pipe1.vcd
From: 0 sec
To: 300 sec
Marker: -- | Cursor: 123 sec
SST
pipe1 test
MYPIPE
Type Signals
wire A[9:0]
wire B[9:0]
wire C[9:0]
wire D[9:0]
wire F[9:0]
reg L12_D[9:0]
reg L12_x1[9:0]
reg L12_x2[9:0]
reg L23_D[9:0]
reg L23_x3[9:0]
reg L34_F[9:0]
wire clk
Filter:
Append
Insert
Replace
Signals
Time
clk
A[9:0]
B[9:0]
C[9:0]
D[9:0]
L12_x1[9:0]
L12_x2[9:0]
L12_D[9:0]
L23_x3[9:0]
L23_D[9:0]
L34_F[9:0]
F[9:0]
Waves
100 sec
200 sec

A Warning

- In this example, we have used a single phase clock to load all the inter-stage registers in the pipeline.
- In a real pipeline, this may lead to race condition.
- Possible solution:
  - Use master-phase flip-flops in the registers.
  - Use a two-phase clock to clock the alternate stages.
- We shall illustrate the two-phase clocking approach in te next example.

END OF LECTURE 33

IIT KHARAGPUR

NPTEL ONLINE
CERTIFICATION COURSES

Lecture 34: PIPELINE MODELING (PART 2)

PROF. INDRANIL SENGUPTA

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

A More Complex Example

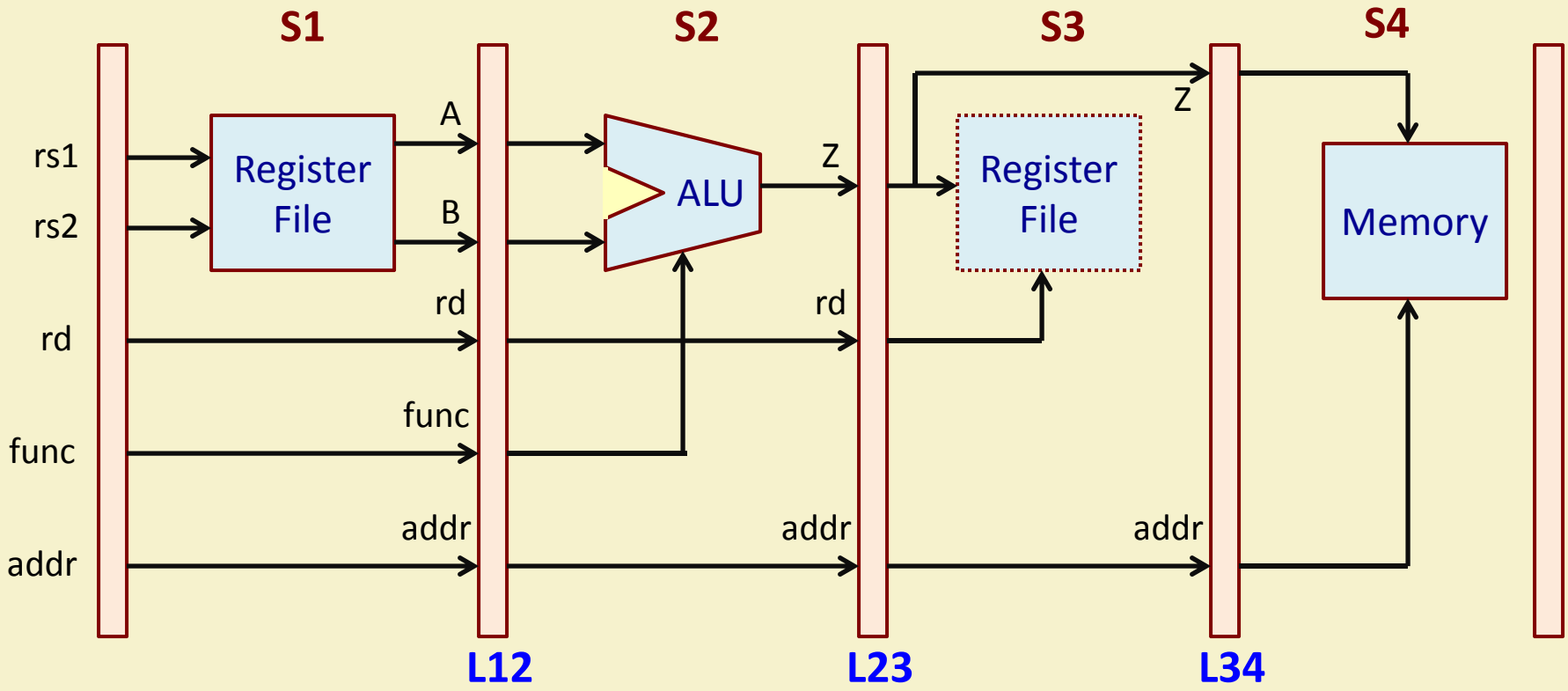
- Consider a pipeline that carries out the following stage-wise operations:
  - *Inputs*: Three register addresses (*rs1*, *rs2* and *rd*), an ALU function (*func*), and a memory address (*addr*).
  - *Stage 1*: Read two 16-bit numbers from the registers specified by "*rs1*" and "*rs2*", and store them in *A* and *B*.
  - *Stage 2*: Perform an ALU operation on *A* and *B* specified by "*func*", and store it in *Z*.
  - *Stage 3*: Write the value of *Z* in the register specified by "*rd*".
  - *Stage 4*: Also write the value of *Z* in memory location "*addr*".

The Assumptions

- There is a register bank containing 16 16-bit registers.
  - 4-bits are required to specify a register address.
  - 2 register reads and 1 register write can be performed every clock cycle.
  - Register addresses are "*rs1*", "*rs2*", and "*rd*".
- Assume that the memory is organized as *256 x 16*.
  - 8-bits are required to specify memory address.
  - Every memory location contains 16 bits of data, which can be read in a single clock cycle.
  - Memory address specified as "*addr*".

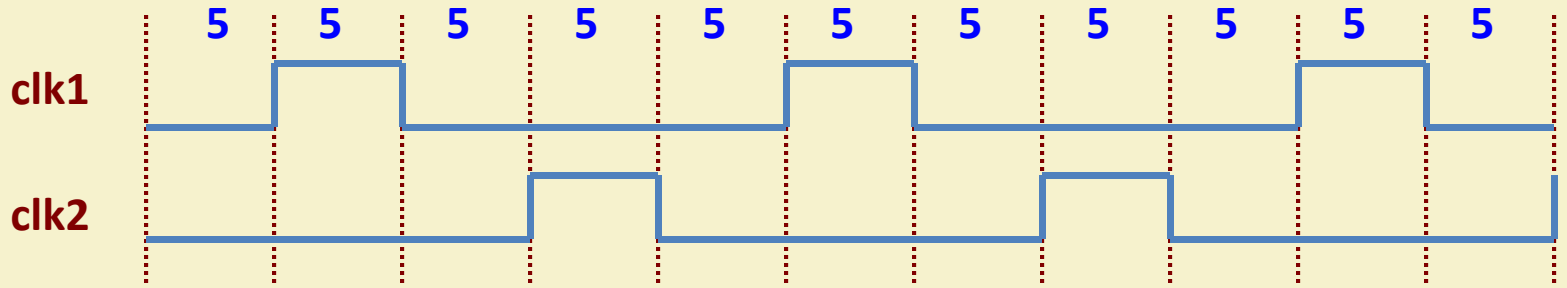
- The ALU function is selected by a 4-bit field "*func*", as follows:

0000: ADD	0001: SUB	0010: MUL
0011: SELA	0100: SELB	0101: AND
0110: OR	0111: XOR	1000: NEGA
1001: NEGB	1010: SRA	1011: SLA


S1
S2
S3
S4
rs1
rs2
rd
func
addr
Register File
A
B
ALU
Z
rd
func
addr
Register File
Z
rd
addr
Memory
addr
L12
L23
L34

Clocking Issue in Pipeline

- It is important that the consecutive stages be applied suitable clocks for correct operation.
- Two options:
  a) Use master/slave flip-flops in the latches to avoid race condition.
  b) Use non-overlapping two-phase clock for the consecutive pipeline stages.



```
module pipe_ex2 (Zout, rs1, rs2, rd, func, addr, clk1, clk2);

  input [3:0] rs1, rs2, rd, func;
  input [7:0] addr;
  input write, clk1, clk2;      // Two-phase clock
  output [15:0] Zout;

  reg [15:0] L12_A, L12_B, L23_Z, L34_Z;
  reg [3:0]  L12_rd, L12_func, L23_rd;
  reg [7:0]  L12_addr, L23_addr, L34_addr;

  reg [15:0] regbank [0:15];  // Register bank
  reg [15:0] mem [0:255];     // 256 x 16 memory

  assign Zout = L34_Z;
```

Pipeline Modeling

```
always @(posedge clk1)
  begin
    L12_A     <=  #2 regbank[rs1];
    L12_B     <=  #2 regbank[rs2];
    L12_rd    <=  #2 rd;
    L12_func  <=  #2 func;
    L12_addr  <=  #2 addr;                  // ** STAGE 1 **
  end
```

```
always @(negedge clk2)
  begin
    case (func)
      0: L23_Z  <=  #2 L12_A + L12_B;
      1: L23_Z  <=  #2 L12_A - L12_B;
      2: L23_Z  <=  #2 L12_A * L12_B;
      3: L23_Z  <=  #2 L12_A;
      4: L23_Z  <=  #2 L12_B;
      5: L23_Z  <=  #2 L12_A & L12_B;
      6: L23_Z  <=  #2 L12_A | L12_B;
      7: L23_Z  <=  #2 L12_A ^ L12_B;
      8: L23_Z  <=  #2 - L12_A;
      9: L23_Z  <=  #2 - L12_B;
      10: L23_Z <=  #2  L12_A >> 1;
      11: L23_Z <=  #2  L12_A << 1;
      default: L23_Z <=  #2 16'hxxxx;
    endcase
    L23_rd    <=  #2 L12_rd;
    L23_addr  <=  #2 L12_addr;          // ** STAGE 2 **
  end
```

```
 always @(posedge clk1)
    begin
      regbank[L23_rd] <=  #2 L23_Z;
      L34_Z     <=  #2 L23_Z;
      L34_addr  <=  #2 L23_addr;            // ** STAGE 3 **
    end

  always @(negedge clk2)
    begin
      mem[L34_addr] <=  #2 L34_Z;           // ** STAGE 4 **
    end

endmodule
```

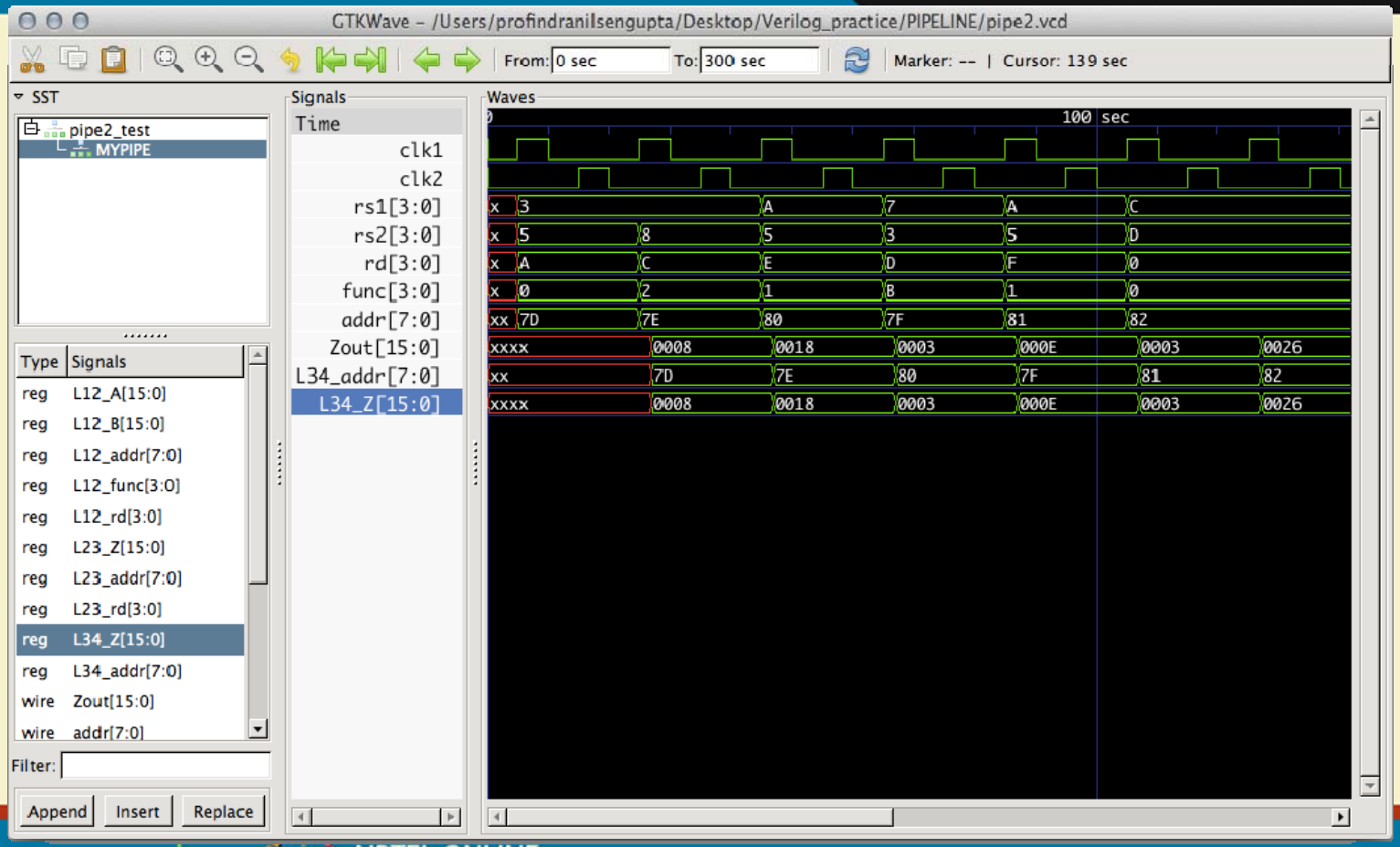
Pipeline Test Bench

```
module pipe2_test;

  wire [15:0] Z;
  reg [3:0] rs1, rs2, rd, func;
  reg [7:0] addr;
  reg clk1, clk2;
  integer k;

  pipe_ex2 MYPIPE (Z, rs1, rs2, rd, func, addr, clk1, clk2);

  initial
    begin
      clk1 = 0; clk2 = 0;
      repeat (20)                  // Generating two-phase clock
        begin
          #5 clk1 = 1;  #5 clk1 = 0;
          #5 clk2 = 1;  #5 clk2 = 0;
        end
     end

  initial
    for (k=0; k<16; k=k+1)
      MYPIPE.regbank[k] = k;       // Initialize registers
```

```
initial
   begin
     #5   rs1 = 3;  rs2 = 5;  rd = 10; func = 0;  addr = 125;  // ADD
     #20  rs1 = 3;  rs2 = 8;  rd = 12; func = 2;  addr = 126;  // MUL
     #20  rs1 = 10; rs2 = 5;  rd = 14; func = 1;  addr = 128;  // SUB
     #20  rs1 = 7;  rs2 = 3;  rd = 13; func = 11; addr = 127;  // SLA
     #20  rs1 = 10; rs2 = 5;  rd = 15; func = 1;  addr = 129;  // SUB
     #20  rs1 = 12; rs2 = 13; rd = 16; func = 0;  addr = 130;  // ADD

     #60 for (k=125; k<131; k=k+1)
       $display ("Mem[%3d] = %3d", k, MYPIPE.mem[k]);
   end

 initial
   begin
     $dumpfile ("pipe2.vcd");
     $dumpvars (0, pipe2_test);
     $monitor ("Time: %3d, F = %3d", $time, Z);
     #300 $finish;
   end

endmodule
```

Simulation Results

```
Time:   0, F =   x
Time:  27, F =   8
Time:  47, F =  24
Time:  67, F =   3
Time:  87, F =  14
Time: 107, F =   3
Time: 127, F =  38
Mem[125] =   8
Mem[126] =  24
Mem[127] =  14
Mem[128] =   3
Mem[129] =   3
Mem[130] =  38
```


GTKWave - /Users/profindranilsengupta/Desktop/Verilog_practice/PIPELINE/pipe2.vcd
From: 0 sec
To: 300 sec
Marker: -- | Cursor: 139 sec
SST
pipe2_test
MYPIPE
Type Signals
reg L12_A[15:0]
reg L12_B[15:0]
reg L12_addr[7:0]
reg L12_func[3:0]
reg L12_rd[3:0]
reg L23_Z[15:0]
reg L23_addr[7:0]
reg L23_rd[3:0]
reg L34_Z[15:0]
reg L34_addr[7:0]
wire Zout[15:0]
wire addr[7:0]
Filter:
Append
Insert
Replace
Signals
Time
clk1
clk2
rs1[3:0]
rs2[3:0]
rd[3:0]
func[3:0]
addr[7:0]
Zout[15:0]
L34_addr[7:0]
L34_Z[15:0]
Waves
100 sec

END OF LECTURE 34

IIT KHARAGPUR

NPTEL ONLINE
CERTIFICATION COURSES

Lecture 35: SWITCH LEVEL MODELING (PART 1)

PROF. INDRANIL SENGUPTA

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Introduction

- A switch level circuit comprises of a netlist of MOS transistors.
- It is not very common for a designer to design modules using transistors.
  - May be required in very specific cases, for designing leaf-level modules in a hierarchical design.
- Verilog provides the ability to model digital circuits at the MOS transistor level.
  - Transistors function as switches; they either conduct (ON) or are open (OFF).
- The four logic levels 0, 1, X, Z and the associated signal drive strengths help in the modeling.
- Two types of switches supported: *ideal* or *resistive*.

Various Switch Primitives in Verilog

- Ideal MOS switches
  - nmos, pmos, cmos
- Resistive MOS switches
  - rnmos, rpmos, rcmos
- Ideal Bidirectional switches
  - tran, tranif0, tranif1
- Resistive Bidirectional switches
  - rtran, rtranif0, rtranif1
- Power and Ground nets
  - supply1, supply0
- Pullup and Pulldown
  - pullup, pulldown

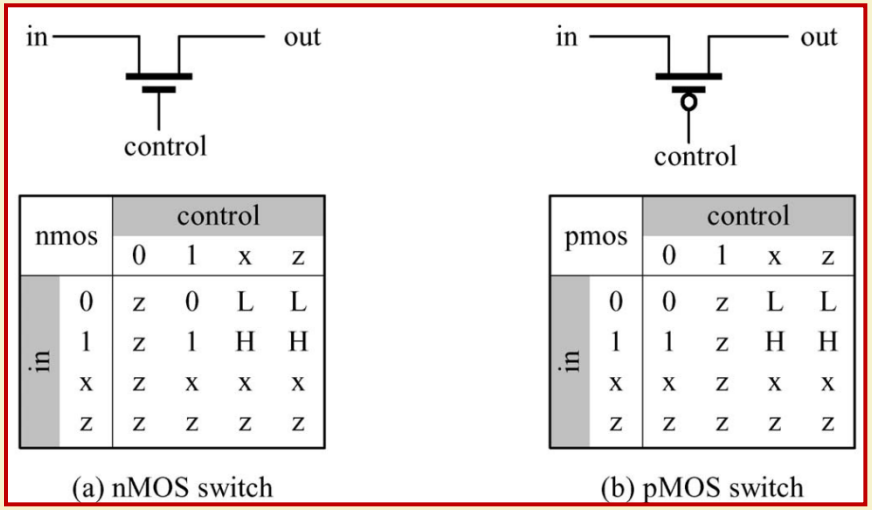
(a) NMOS and PMOS Switches

- Declared with keywords "*nmos*" and "*pmos*".
- Format for instantiation:

```
nmos (or pmos) [instance_name] (output, input, control);
```

Here, "instance_name" is optional.

Also called pass transistors.



nmos		control			
		0	1	x	z
in	0	z	0	L	L
	1	z	1	H	H
	x	z	x	x	x
	z	z	z	z	z

(a) nMOS switch

pmos		control			
		0	1	x	z
in	0	0	z	L	L
	1	1	z	H	H
	x	x	z	x	x
	z	z	z	z	z

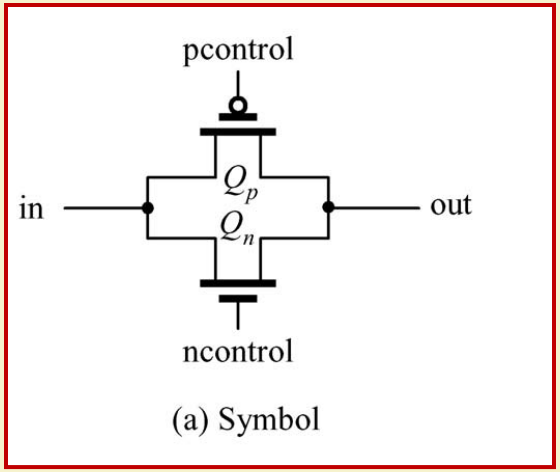
(b) pMOS switch

(b) CMOS Switch (Transmission Gate)

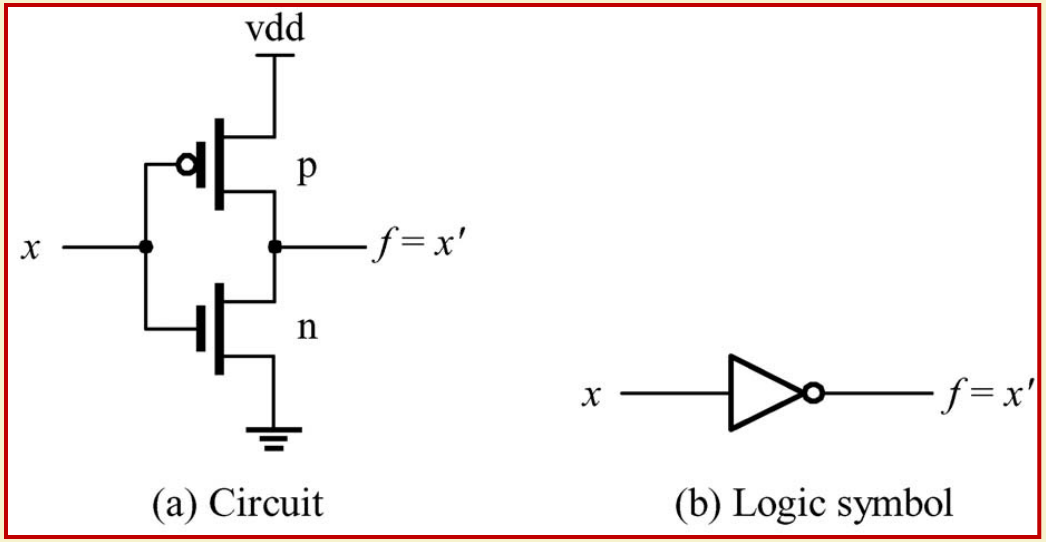
- Declared with keywords "*cmos*".
- Format for instantiation:

```
cmos [instance_name] (output, input, ncontrol, pcontrol);
```

Here also, "instance_name" is optional.

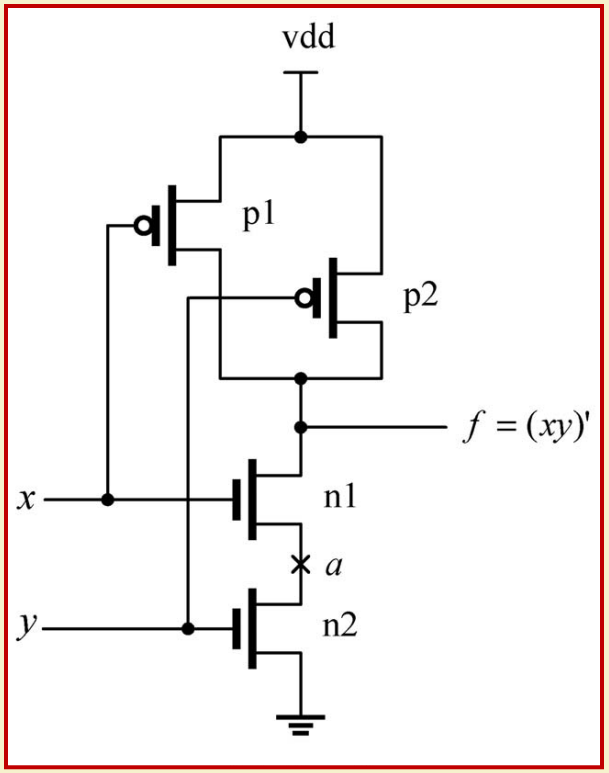


Example 1: CMOS Inverter




```
module cmosnot (x, f);
  input x;
  output f;
  supply1 vdd;
  supply0 gnd;
  pmos p1 (f, vdd, x);
  nmos n1 (f, gnd, x);
endmodule
```

Example 2: CMOS NAND Gate

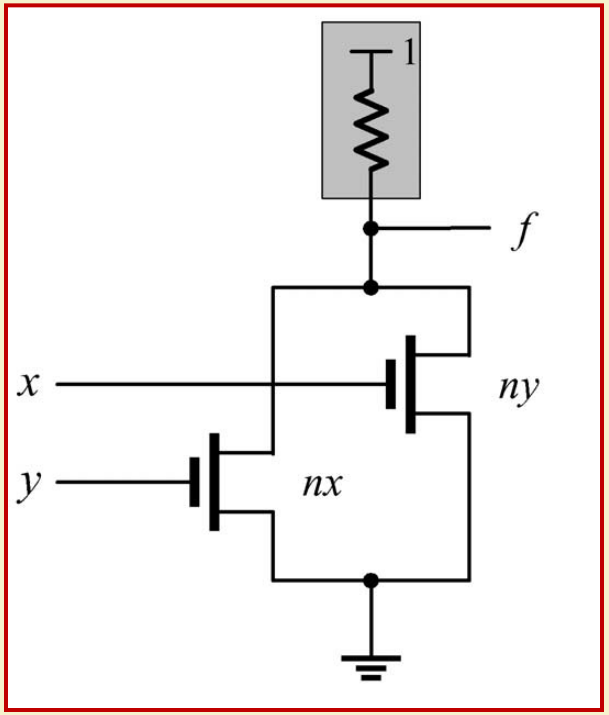



```
module cmosnand (x, y, f);
  input x, y;
  output f;
  supply1 vdd;
  supply0 gnd;
  wire a;
  pmos p1 (f, vdd, x);
  pmos p2 (f, vdd, y);
  nmos n1 (f, a, x);
  nmos n2 (a, gnd, y);
endmodule
```

```
module cmosnand_test;

  reg in1, in2;
  wire out;
  integer k;

  cmosnand MYNAND2 (in1, in2, out);

  initial
    begin
      for (k=0; k<4; k=k+1)
        begin
          #5 {in1,in2} = k;
          $display ("In1: %b, In2: %b, Out: %b", in1, in2,
out);
        end
    end

endmodule
```

```
In1: 0, In2: 0, Out: 1
In1: 0, In2: 1, Out: 1
In1: 1, In2: 0, Out: 1
In1: 1, In2: 1, Out: 0
```

Example 3: Pseudo-NMOS NOR Gate

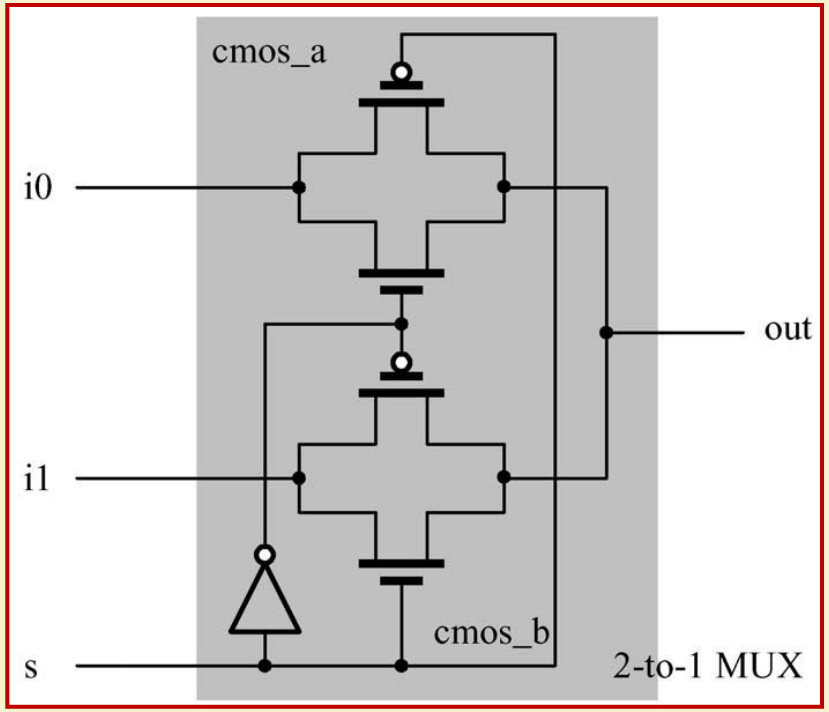



```
module pseudonor (x, y, f);
  input x, y;
  output f;
  supply0 gnd;
  nmos nx (f, gnd, x);
  nmos ny (f, gnd, y);
  pullup (f);
endmodule
```

```
module pseudonor_test;

  reg in1, in2;
  wire out;
  integer k;
  pseudonor MYNOR2 (in1, in2, out);
  initial
    begin
      for (k=0; k<4; k=k+1)
        begin
          #5 {in1,in2} = k;
          $display ("In1: %b, In2: %b, Out: %b", in1, in2,
out);
        end
    end
endmodule
```

```
In1: 0, In2: 0, Out: 1
In1: 0, In2: 1, Out: 0
In1: 1, In2: 0, Out: 0
In1: 1, In2: 1, Out: 0
```

Example 4: CMOS 2x1 Multiplexer



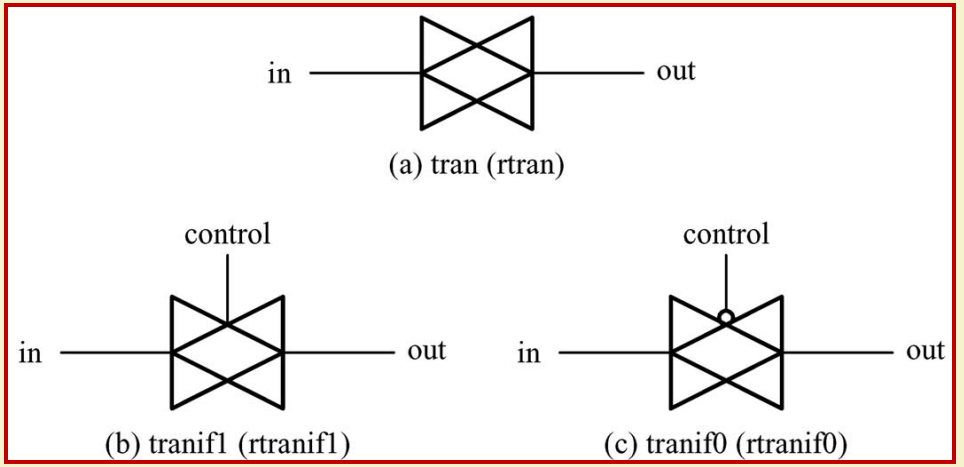

```
module mux_2to1 (out, s, i0, i1);
  input s, i0, i1;
  output out;
  wire sbar;
  not (sbar, s);
  cmos cmos_a (out, i0, sbar, s);
  cmos cmos_b (out, i1, s, sbar);
endmodule
```

```
module cmosmux_test;
  reg sel, in0, in1;
  wire out;
  integer k;
  cmosmux MUX21 (out, sel, in0, in1);
  initial
    begin
      for (k=0; k<8; k=k+1)
        begin
          #5 {sel,in0,in1} = k;
          $display ("Sel: %b, In0: %b, In1: %b, Out: %b",
                                          sel, in0, in1,
out);
        end
    end
endmodule
```

```
Sel: 0, In0: 0, In1: 0, Out: 0
Sel: 0, In0: 0, In1: 1, Out: 0
Sel: 0, In0: 1, In1: 0, Out: 1
Sel: 0, In0: 1, In1: 1, Out: 1
Sel: 1, In0: 0, In1: 0, Out: 0
Sel: 1, In0: 0, In1: 1, Out: 1
Sel: 1, In0: 1, In1: 0, Out: 0
Sel: 1, In0: 1, In1: 1, Out: 1
```

(c) Bidirectional Switches

- "*nmos*", "*pmos*", and "*cmos*" gates conduct in one direction (drain to source).
- When it is required for devices to conduct in both direction, we use bidirectional switches.
- Three types of bidirectional switches: "*tran*", "*tranif0*", and "*tranif1*".



Syntax:

```
tran    [instance_name] (inout1, inout2);
tranif0 [instance_name] (inout1, inout2, cntl);
tranif1 [instance_name] (inout1, inout2, cntl);
```

END OF LECTURE 35

IIT KHARAGPUR

NPTEL ONLINE
CERTIFICATION COURSES

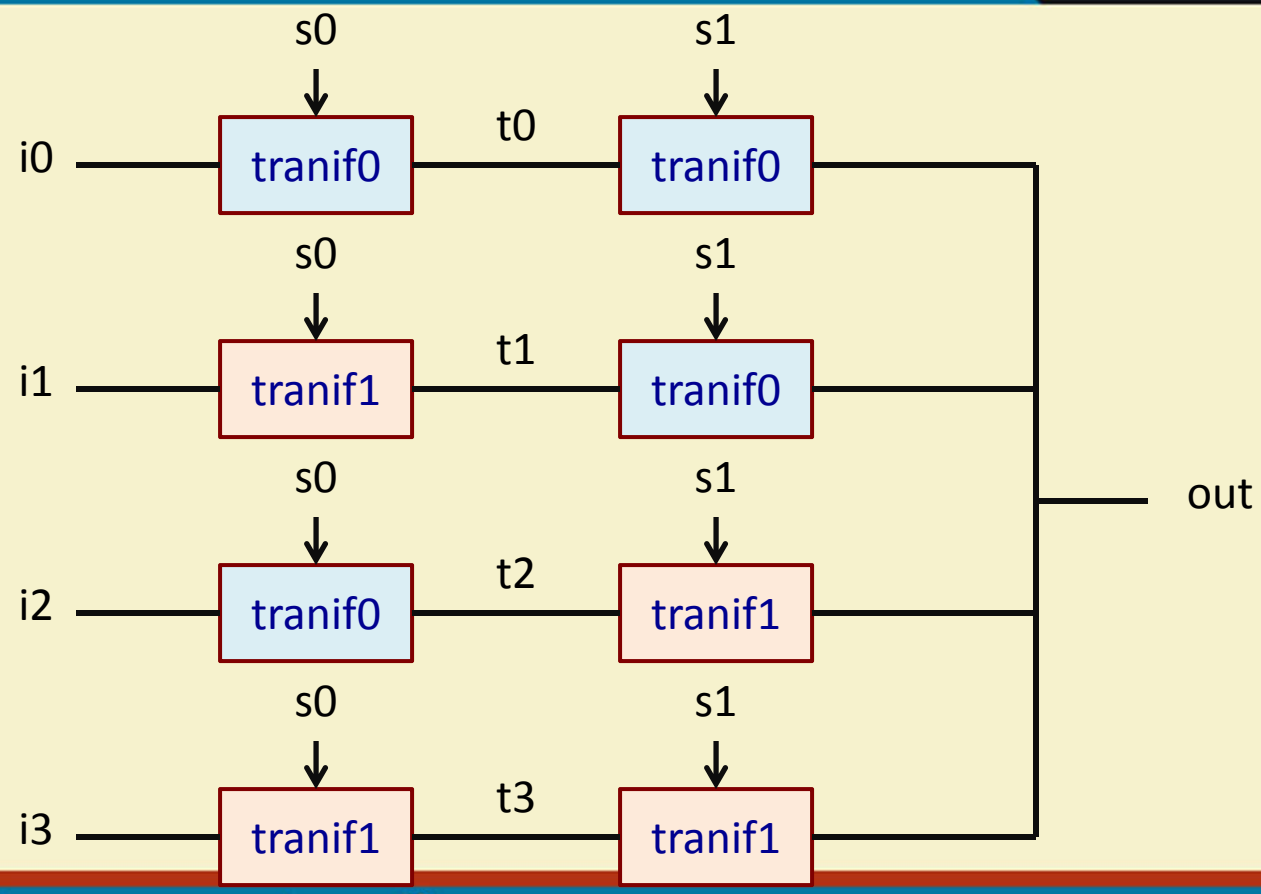
Lecture 36: SWITCH LEVEL MODELING (PART 2)

PROF. INDRANIL SENGUPTA

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Example 5: CMOS 4x1 Multiplexer using “tran” Switches

```
module mux_4to1 (out, s0, s1, i0, i1, i2, i3);
  input s0, s1, i0, i1, i2, i3;
  output out;
  wire t0, t1, t2, t3;

  tranif0 (i0, t0, s0);     tranif0 (t0, out, s1);
  tranif1 (i1, t1, s0);     tranif0 (t1, out, s1);
  tranif0 (i2, t2, s0);     tranif1 (t2, out, s1);
  tranif1 (i3, t3, s0);     tranif1 (t3, out, s1);
endmodule
```



Schematic Diagram

Test Bench

```
module mux41_test;
  reg s0, s1, i0, i1, i2, i3;
  wire out;
  integer k;
  mux_4to1 MYMUX41 (out, s0, s1, i0, i1, i2, i3);
  initial
    begin
      for (k=0; k<64; k=k+1)
        begin
          #5 {s0,s1,i0,i1,i2,i3} = k;
          $display ("Sel: %2b, In: %4b, Out: %b",
                                  {s0,s1}, {i0,i1,i2,i3}, out);
        end
    end
endmodule
```

Simulation Results

```
Sel: 00, In: 0000, Out: x
Sel: 00, In: 0001, Out: 0
Sel: 00, In: 0010, Out: 0
Sel: 00, In: 0011, Out: 0
Sel: 00, In: 0100, Out: 0
Sel: 00, In: 0101, Out: 0
Sel: 00, In: 0110, Out: 0
Sel: 00, In: 0111, Out: 0
Sel: 00, In: 1000, Out: 0
Sel: 00, In: 1001, Out: 1
Sel: 00, In: 1010, Out: 1
Sel: 00, In: 1011, Out: 1
Sel: 00, In: 1100, Out: 1
Sel: 00, In: 1101, Out: 1
Sel: 00, In: 1110, Out: 1
Sel: 00, In: 1111, Out: 1
```

```
Sel: 01, In: 0000, Out: 1
Sel: 01, In: 0001, Out: 0
Sel: 01, In: 0010, Out: 0
Sel: 01, In: 0011, Out: 1
Sel: 01, In: 0100, Out: 1
Sel: 01, In: 0101, Out: 0
Sel: 01, In: 0110, Out: 0
Sel: 01, In: 0111, Out: 1
Sel: 01, In: 1000, Out: 1
Sel: 01, In: 1001, Out: 0
Sel: 01, In: 1010, Out: 0
Sel: 01, In: 1011, Out: 1
Sel: 01, In: 1100, Out: 1
Sel: 01, In: 1101, Out: 0
Sel: 01, In: 1110, Out: 0
Sel: 01, In: 1111, Out: 1
```

Simulation Results

```
Sel: 10, In: 0000, Out: 1
Sel: 10, In: 0001, Out: 0
Sel: 10, In: 0010, Out: 0
Sel: 10, In: 0011, Out: 0
Sel: 10, In: 0100, Out: 0
Sel: 10, In: 0101, Out: 1
Sel: 10, In: 0110, Out: 1
Sel: 10, In: 0111, Out: 1
Sel: 10, In: 1000, Out: 1
Sel: 10, In: 1001, Out: 0
Sel: 10, In: 1010, Out: 0
Sel: 10, In: 1011, Out: 0
Sel: 10, In: 1100, Out: 0
Sel: 10, In: 1101, Out: 1
Sel: 10, In: 1110, Out: 1
Sel: 10, In: 1111, Out: 1
```

```
Sel: 11, In: 0000, Out: 1
Sel: 11, In: 0001, Out: 0
Sel: 11, In: 0010, Out: 1
Sel: 11, In: 0011, Out: 0
Sel: 11, In: 0100, Out: 1
Sel: 11, In: 0101, Out: 0
Sel: 11, In: 0110, Out: 1
Sel: 11, In: 0111, Out: 0
Sel: 11, In: 1000, Out: 1
Sel: 11, In: 1001, Out: 0
Sel: 11, In: 1010, Out: 1
Sel: 11, In: 1011, Out: 0
Sel: 11, In: 1100, Out: 1
Sel: 11, In: 1101, Out: 0
Sel: 11, In: 1110, Out: 1
Sel: 11, In: 1111, Out: 0
```

Example 6: Full Adder using Transistor Level Modeling

```
module fulladder (sum, cout, a, b, cin);
  input a, b, cin;
  output sum, cout;
  fa_sum SUM (sum, a, b, cin);
  fa_carry CARRY (cout, a, b, cin);
endmodule
```

```
module fa_carry (cout, a, b, cin);
  input a, b, cin;
  output cout;
  wire t1, t2, t3, t4, t5;
  cmosnand N1 (t1, a, b);
  cmosnand N2 (t2, a, cin);
  cmosnand N3 (t3, b, cin);
  cmosnand N4 (t4, t1, t2);
  cmosnand N5 (t5, t4, t4);
  cmosnand N6 (cout, t5, t3);
endmodule
```

```
module myxor2 (out, a, b);
  input a, b;
  output out;
  wire t1, t2, t3, t4;

  cmosnand N1 (t1, a, a);
  cmosnand N2 (t2, b, b);
  cmosnand N3 (t3, a, t2);
  cmosnand N4 (t4, b, t1);
  cmosnand N5 (out, t3, t4);
endmodule
```

```
module fa_sum (sum, a, b, cin);
  input a, b, cin;
  output sum;
  wire t1, t2;

  myxor2 X1 (t1, a, b);
  myxor2 X2 (sum, t1, cin);
endmodule
```

```
module cmosnand (f, x, y);
  input x, y;
  output f;
  supply1 vdd;
  supply0 gnd;
  pmos p1 (f, vdd, x);
  pmos p2 (f, vdd, y);
  nmos n1 (f, a, x);
  nmos n2 (a, gnd, y);
endmodule
```

Test Bench

```
module fulladder_test;
  reg a, b, cin;
  wire sum, cout;
  integer k;
  fulladder FA (sum, cout, a, b, cin);
  initial
    begin
      for (k=0; k<8; k=k+1)
        begin
          #5 {a, b, cin} = k;
          $display ("Inputs: %3b Sum: %b, Carry: %b",
                                          {a,b,cin}, sum, cout);
        end
    end
endmodule
```

Simulation Results

```
Inputs: 000 Sum: 0, Carry: 0
Inputs: 001 Sum: 1, Carry: 0
Inputs: 010 Sum: 1, Carry: 0
Inputs: 011 Sum: 0, Carry: 1
Inputs: 100 Sum: 1, Carry: 0
Inputs: 101 Sum: 0, Carry: 1
Inputs: 110 Sum: 0, Carry: 1
Inputs: 111 Sum: 1, Carry: 1
```

Data Values and Signal Strengths

- Verilog supports 4 value levels and 8 strength levels to model the functionality of real hardware.
  - Strength levels are typically used to resolve conflicts between signal drivers of different strengths in real circuits.

Value Level	Represents
0	Logic 0 state
1	Logic 1 state
x	Unknown logic state
z	High impedance state

Initialization:

- All unconnected nets are set to “z”.
- All register variables set to “x”.

Strength	Type
supply	Driving
strong	Driving
pull	Driving
large	Storage
weak	Driving
medium	Storage
small	Storage
highz	High impedance

- If two signals of unequal strengths get driven on a wire, the stronger signal will prevail.
- These are particularly useful for MOS level circuits, e.g. dynamic MOS.
- When a signal passes through a resistive switch, for example, its strength reduces.
- There is a convention followed for signal strength computation in MOS level circuits.
  - Details not discussed here.

Point to Note

- Most of the synthesis tools do not support switch level modeling.
  - Because it uses technology mapping from a given library.
  - Common gates and simple functional blocks are present in the library.
  - Thus it is not required to specify circuits at the transistor level.

END OF LECTURE 36